Increasing Assurance with Literate Programming Techniques

Andrew P. Moore
Code 5542
Naval Research Laboratory
Washington, DC 20375
moore@itd.nrl.navy.mil

Charles N. Payne, Jr.*
Secure Computing Corporation
2675 Long Lake Road
Roseville, MN 55113
cpayne@sctc.com

Abstract

The assurance argument that a trusted system satisfies its information security requirements must be convincing, because the argument supports the accreditation decision to allow the computer to process classified information in an operational environment. Assurance is achieved through understanding, but some evidence that supports the assurance argument can be difficult to understand. This paper describes a novel application of a technique, called literate programming [11], that significantly improves the readability of the assurance argument while maintaining its consistency with formal specifications that are input to specification and verification systems. We describe an application of this technique to a simple example and discuss the lessons learned from this effort.

1 Introduction

A convincing argument, called an *assurance argument*, must be made that a trusted system is trustworthy.[1] The assurance argument supports the accreditation decision to allow a computer to process classified information in an operational environment. The argument is composed from technical evidence that is produced during the system development process, e.g., the security model, design specifications, proofs, vulnerability analysis and test results. As the trust in a system increases, the demand for rigor in the system's assurance argument also increases [5, 23]. However, increasing the rigor of an argument does not necessarily make it more convincing.

Assurance is achieved through understanding, specifically the understanding of parties *independent* of the development effort, i.e., the certifiers. Rigorously produced evidence that is difficult to understand contributes little to assurance [19]. Ricky Butler notes that formal methods can help us demonstrate that an implementation satisfies its specification, but ultimately we must validate that the specification describes what was intended [4]. The assurance argument must help the certifier understand why the specification is accurate (validation) and why the implementation satisfies the specification (verification). The success of an assurance argument, therefore, depends as much on its *presentation* as on its *production.*

We have developed and demonstrated an assurance argument documentation approach that improves the readability of the assurance argument while maintaining its consistency with formal specifications that are input to specification and verification systems. Our approach relies on a new application of old technology. Over a decade ago, Donald Knuth introduced the concept of literate programming (LP) for improving the readability and comprehension of computer programs [11]. He believes that programs should be written to be read by humans instead of computers. Since then, others have extended the application of LP techniques to program design languages [3], formal specification languages [32], and formal proofs [30]. We propose here that LP techniques can be used to document the *entire* assurance argument.

This paper describes our approach for documenting a formal assurance argument using LP techniques. In the next section, we discuss some problems that may be encountered when documenting a formal assurance argument. Then, we review the LP paradigm in the context of available tools and describe how we used LP techniques to document an assurance argument for a small example. We conclude with some lessons learned from that effort and an overview of the primary benefit that accrues from using our approach. An appendix to the paper presents an excerpt from the assurance argument for the example described.

2 Understanding the Problem

A few years ago we constructed a formal assurance argument for a highly trusted network security device [25]. Our chain of reasoning spanned from the critical

*Work performed while author was employed by the Naval Research Laboratory.

[1] A computer system is *trusted* if we rely on it for security enforcement. It is *trustworthy* if that reliance is justified technically. A computer may be trusted even though it is not trustworthy, because the accreditor may permit its use despite known weaknesses.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **1996**	2. REPORT TYPE	3. DATES COVERED **00-00-1996 to 00-00-1996**
4. TITLE AND SUBTITLE **Increasing Assurance with Literate Programming Techniques**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Code 5542,4555 Overlook Avenue, SW,Washington,DC,20375**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**12**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

requirements model to source code that was verified to conform to the model. The argument was constructed using a specification language for composite (concurrent) systems, a specification language and verification system for the refinement of sequential components, and a sequential programming language. Assurance evidence produced included machine-generated and manual proofs, a hand-translation from the composite system specification to the sequential component specification, a hand-translation from the sequential component specification to the programming language, and rigorous functional testing. In addition, we modeled graphically the functional behavior of the device. We presented each piece of evidence to the certifier, and we described verbally the connections between them.

Unfortunately, the rigor of the assurance argument for the network security device did not significantly improve the certifier's confidence that the critical requirements model was accurate nor that the system, as implemented, conformed to the model. Much of the argument's power to persuade was lost for the following reasons:

- *The framework for the assurance argument was not documented adequately.*

  An accurate understanding of the argument requires that the framework specify underlying assumptions, motivations and strategies within the context of the system's specification. Significant results of the argument need to be highlighted, and meaningful threads between different pieces of evidence need to be traced. The framework must clearly specify how each piece of evidence contributes to the overall argument. This is particularly important in the development of non-trivial systems that rely on several formal methods, each of which makes certain (possibly contradictory) assumptions about the system being specified. A good framework ensures the cohesion of the argument.

- *The specifications and proofs were documented in a manner to facilitate acceptance by mechanical tools rather than to promote understanding by humans.*

  Since the mechanical tools lack many of the reasoning and inference capabilities of a human reader, more information was needed to facilitate verification, e.g., proof heuristics. This extra information can be a substantial obstacle to comprehension. In addition, computer language compilers often restrict the ordering of the individual parts of the specification, e.g., requiring that subroutines be defined before they are called. Unfortunately, the order required by the compiler may not be the most intuitive for the human reviewer. Methods are needed to help manage specification detail by guiding the reader through the specification, pointing out areas of particular interest and hiding obvious or extraneous detail.

- *Little assurance was provided that the documentation accurately represents the system as implemented.*

  The assurance argument for the network security device was partitioned into system requirements, module interfaces and module implementation documentation. Documents were kept consistent with the formal specification and code manually; any change to the specification or code required a change to the documentation. This approach is both labor-intensive and error-prone and does not promote confidence in the running system. Any certification of a system must be based on documentation that incontrovertibly describes the system executables.

The problems with certifying the network security device suggested the need for tools and techniques to improve the presentation of an assurance argument without sacrificing its rigor.

3 Literate Programming (LP)

To demonstrate the literate programming concepts, Knuth developed a toolset called `WEB` to support writing Pascal programs in the literate style. A `WEB` literate program is a text file that contains both computer code and descriptive narrative. The structure of the file is determined by the author instead of the compiler. The code can be broken into small, understandable chunks (which we call macros) and documented in any order. `WEB` tools process text files in two independent phases. In the *tangle* phase, the computer code is extracted and re-ordered for the programming language compiler. In the *weave* phase, the macros are cross-referenced, program variables are indexed and typesetting commands are added to the code and to the descriptive narrative. Figure 1 illustrates that the results of these phases, after processing by a compiler and a typesetter, are executable code and typeset documentation, respectively. The figure also illustrates a unique characteristic of the LP approach: because there is only *one* source file, the code that is documented is always consistent with the code that is executed. From a certifier's perspective, this is valuable assurance evidence.

The literate program development lifecycle (see Figure 2) is not that different from the conventional program development lifecycle. With conventional program development, if there are errors in the *compile* or *run* phase, the developer returns to the *edit* phase and modifies the source code. Literate program development introduces the *tangle* phase. The output of the *edit* phase is a literate program rather than compiler-ready source code. If an error occurs in the *compile* or *run* phases, the developer must modify the literate program (instead of the source code). Knuth discouraged the programmer from editing the source code by making the output of the *tangle* phase almost unreadable.

Very simple cross-referencing information is added to

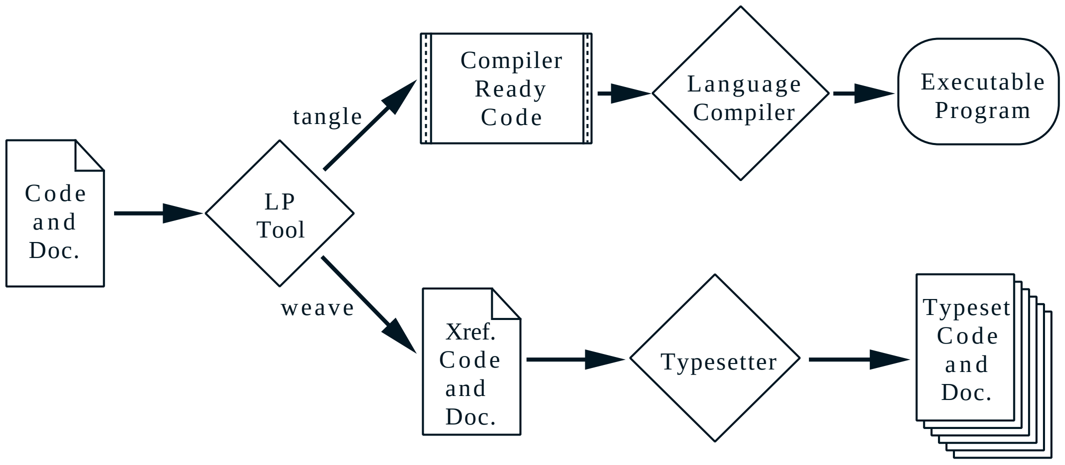


Figure 1: The `WEB` process

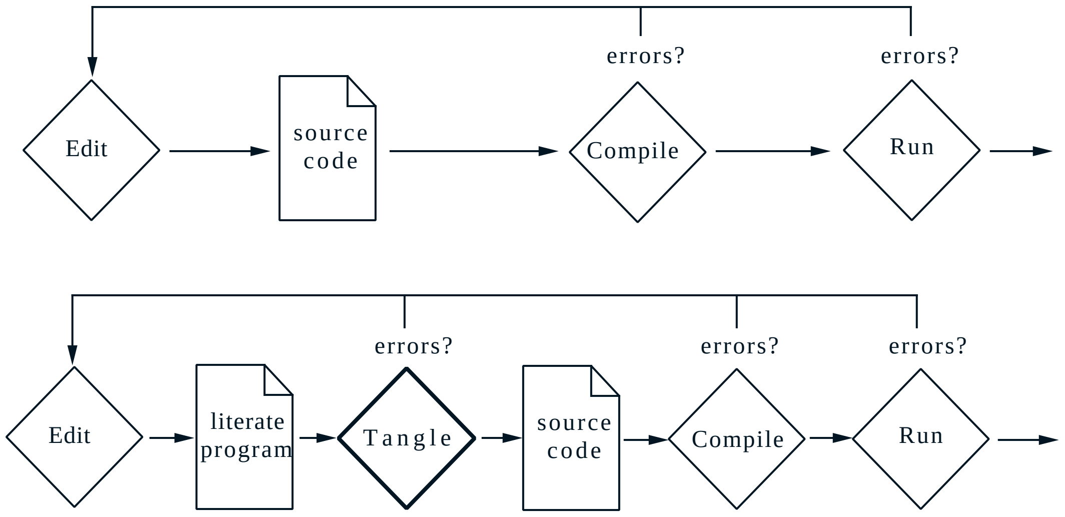
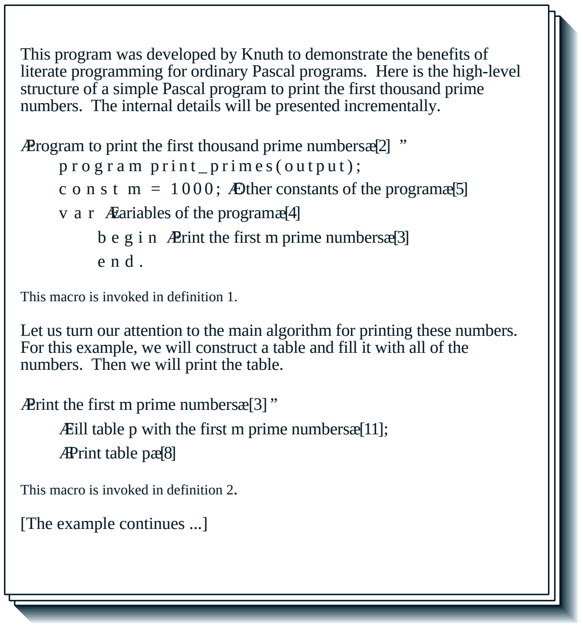


Figure 2: Conventional program development (top) and literate program development (bottom)

macros during the *weave* phase. An example is illustrated in Figure 3, where two macros, “Program to print the first thousand prime numbers” and “Print the first m prime numbers”, are defined [12].[2] Each macro is assigned a unique number (in this case, 2 and 3) during the *weave* phase. The macro body consists of code and references to other macros. For example, macro 2 includes a reference to macro 3. During the *tangle* phase, these references are expanded beginning with the “root macro” (not pictured) until all code is revealed. Macros need not be defined in the order that they are referenced. The example illustrated in Figure 3 could be refined to discuss how to print table p before discussing how to fill table p with prime values.



This program was developed by Knuth to demonstrate the benefits of literate programming for ordinary Pascal programs. Here is the high-level structure of a simple Pascal program to print the first thousand prime numbers. The internal details will be presented incrementally.

```
ÆProgram to print the first thousand prime numbersæ[2] ”
    p r o g r a m  p r i n t _ p r i m e s ( o u t p u t ) ;
    c o n s t  m  =  1 0 0 0 ;  ÆOther constants of the programæ[5]
    v a r  ÆVariables of the programæ[4]
        b e g i n  ÆPrint the first m prime numbersæ[3]
        e n d .
```

This macro is invoked in definition 1.

Let us turn our attention to the main algorithm for printing these numbers. For this example, we will construct a table and fill it with all of the numbers. Then we will print the table.

```
ÆPrint the first m prime numbersæ[3] ”
    ÆFill table p with the first m prime numbersæ[11];
    ÆPrint table pæ[8]
```

This macro is invoked in definition 2.

[The example continues ...]

Figure 3: Sample LP macros

Knuth notes several benefits of LP [14]:

- There is no need to choose between top-down and bottom-up programming, because a program is treated as a web instead of a tree. A hierarchical structure is present, but the most important characteristic of the program is its structural relationships. A complex piece of software consists of simple parts and simple relationships between these parts. The programmer’s task is to state those parts and their relationships in whatever order is best for the reader.
- Each part of a program can have its appropriate stature without distorting the readability of other parts.
- A literate program takes less time to debug because the programmer is encouraged to clarify his thoughts as he programs.

[2]Note that the numbers in square brackets in the figure are not bibliographic references, but macro definition numbers. These numbers appear only after macro definitions or references, i.e., the angle bracket delimited items, and so should not cause confusion with numbers in square brackets that denote bibliographic entries elsewhere in the document.

The popularity of `WEB` for Pascal motivated the development of LP tools for other languages, e.g., C [18] Smalltalk [27], and Modula-2 [29]. However, it soon became apparent that it would be impractical to build a separate tool for every programming language in use. In addition, users demanded tools that could process more than one language simultaneously. This motivated the development of language-independent tools such as nuweb [2], noweb [26] and FunnelWeb [31]. Not only can these tools handle multiple languages, but they are not limited to computer programming languages. For this flexibility, language-independent tools sacrifice features such as code prettyprinting and automatic indexing of program variables.

4 A Simple Application of LP

We chose to investigate the use of LP techniques to improve the presentation of rigorous assurance arguments on a relatively simple problem, an RS-232 character repeater [17]. Choosing a simple problem allows us to observe the influence that LP has on our specification and verification styles without getting bogged down by the details of the problem. This section describes the character repeater problem and how we documented it. The repeater assurance argument [22] presents further details about the problem and our solution.

The character repeater, illustrated in Figure 4, relays only even parity characters until it overflows. More specifically, the repeater has an input port, an output port, and an error port that can operate at a range of speeds. Characters of even parity received over the input port are stored in an internal buffer prior to their transmission over the output port. Characters of odd parity cause an error to be signaled over the error port and are discarded. If the internal buffer overflows, an error is signaled to the error port, all characters in the buffer are transmitted, and the repeater halts.

The repeater must satisfy the following critical requirements:

- Exactly those characters of even parity received by the repeater prior to the reception of the character causing an overflow are transmitted.
- Only received characters are transmitted.
- Characters are transmitted in the order in which they were received.

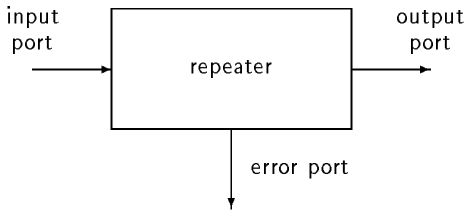


Figure 4: The RS-232 character repeater

The development and documentation of the repeater assurance argument involved the use of a number of languages and tools. The CSP language [10] is the computational framework for the repeater specification and verification. The EVES Verification System [6, 15] and the FDR model checker [7, 28] provide mechanical assistance for constructing and verifying the CSP specifications. EVES is a general purpose interactive verification system that can be used to prove mathematically that CSP descriptions conform to trace specifications [21, 24]. FDR is a CSP model checker that automatically verifies (through an exhaustive state space analysis) that a CSP process implementation properly refines a CSP process specification. Finally, FunnelWeb [31] is the LP tool that is used to facilitate the documentation of the repeater assurance argument. FunnelWeb allows documenting both the EVES and FDR specifications simultaneously by allowing one to tangle multiple output files.

Figure 5 illustrates the structure of the assurance argument for the repeater CSP process implementation. Slanted arrows indicate a *refinement* of a specification to a more detailed specification or implementation; vertical arrows indicate a *translation* of a specification from one semantic domain to a comparable semantic domain. Dashed arrows indicate a refinement/translation that is *informal*; solid arrows indicate a refinement that uses a combination of *informal* and *formal* techniques. The increase in width of the argument from top to bottom illustrates additional detail that is specified at the lower levels. The translation in the middle of the figure is necessary because FDR lacks the power to analyze the top-level critical requirements directly. We used EVES to refine the top-level critical requirements to a semantic level that FDR can handle; then we translated the specification into FDR's syntax.

The repeater assurance argument was successfully completed. It is documented as a single coherent entity, from the top-level problem description to the detailed physical design[22]. It describes in detail not only the major components of the assurance argument shown in Figure 5, but also the transition between these components. Figure 6 illustrates how FunnelWeb was used to document the argument. FunnelWeb allows tangling both EVES specification files (those ending with "sv") and FDR specification files (those ending with

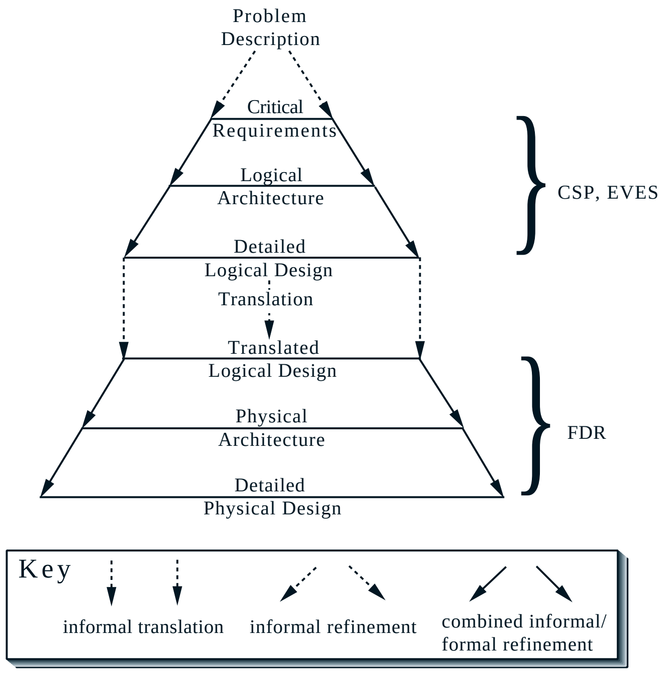


Figure 5: Repeater Assurance Argument

"fdr"). The final results of the FDR and EVES analysis and processing are documented in the assurance argument. The argument is woven into a source file (that ending with "tex") for processing using LaTeX [16], a macro language for TeX [13] that has powerful cross-referencing capabilities. For simplicity, the figure does not represent the interactive and iterative nature of the process of developing the assurance argument. An excerpt from the argument, shown in the appendix, formalizes the repeater's critical requirements. This excerpt illustrates the general flow of the LP presentation of the argument, even if the reader is unfamiliar with the EVES and CSP languages used.

The repeater problem has several qualities that made it a good choice for our investigation: it is simple, self-contained and well-understood. This allowed us to concentrate on the use of LP techniques to develop and explicate its assurance argument with minimal overhead in actually implementing the repeater itself. Unfortunately, it makes it more difficult to argue that the techniques scale-up to larger, more complex application. Nevertheless, the refinement of the repeater addresses many important issues that commonly arise in the development of more complex hardware/software systems:

- **Stating Requirements Intuitively:** The repeater's critical requirements are formalized as restrictions on the input/output behavior (traces) of the repeater viewed as a black-box. This abstraction permits a more intuitive formulation of the

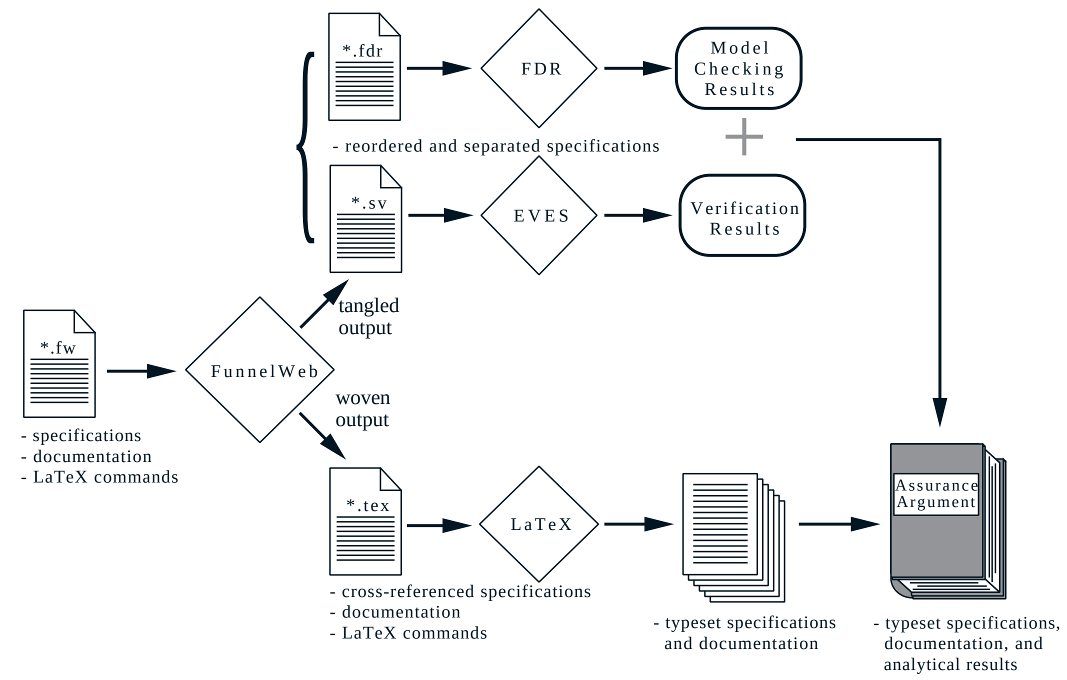


Figure 6: Documenting the repeater assurance argument

repeater's actual requirements than that allowed by many verification systems, especially symbolic model checkers.

- **Dealing Explicitly with Concurrency:** The repeater is specified as a composition of communicating, sequential processes executing concurrently. A method described in [20] was used to derive requirements for the sequential processes sufficient to guarantee that the composite process satisfies its specification. Dealing with concurrency explicitly allowed us to uncover potential problems with the implementation (e.g., synchronization problems) that models based on a single sequential state machine might gloss over.

- **Specifying Both the Logical and Physical Design:** The decomposition of the repeater into communicating sequential processes was first specified and verified as a logical design (in EVES) and then translated to a physical design (in FDR). The logical design describes a simple abstraction of the repeater (as two processes executing in parallel) that is shown to enforce trace specifications of its critical requirements using EVES. The physical design implements the abstract specification as three processes executing concurrently; it is shown to conform to the logical design using FDR.[3] Being able to specify multiple levels of abstraction in the design process is critical to managing the verification of complex systems.

By explicitly addressing these three issues we demonstrate the potential scalability of our approach.

The resulting assurance argument is 130 pages, divided into four parts containing thirteen chapters, with three appendices, an index and a bibliography. The four parts are the repeater's problem statement, the logical design, the physical design, and supporting definitions. Each part contains one or more chapters. For example, Part IV contains several chapters consisting of selected definitions from the underlying theories of the specification. The only definitions presented are those considered necessary to understand the specification. The theories from which the definitions are extracted are defined in separate documents that accompany the argument.

The length and complexity of the argument is primarily due to our goal of addressing issues that would have to be dealt with in larger-scale applications, as described above. Nevertheless, the length of the assurance ev-

[3]A collaborative effort with George Mason University [1] yielded a verified gate-level implementation of the physical design.

idence can be intimidating. The EVES specification and its supporting theories span ninety pages not including the narrative description. Fortunately, only a small fraction of this information needs to be understood to comprehend the assurance argument, and LP tools permit extracting just those elements.

5 Lessons Learned

Our trial application of LP techniques to the RS-232 repeater assurance argument suggests several lessons to keep in mind when applying the approach in the future:

Adopt tutorial application of LP to ease certification.

Documentation that successfully supports system development and maintenance must allow users to locate efficiently the elements of the system specification that they need. Documentation that successfully supports certification must convince readers that the system satisfies its critical requirements. Unfortunately, conventional approaches to documenting systems often requires one to sacrifice ease of understanding for ease of reference.

However, when applied to system specifications, LP tools and techniques permit the user to construct two views of the specification - one produced as a result of the weave phase and one as a result of the tangle phase. The literate program produced as a result of the weave phase should be written as a tutorial explication of the assurance argument to facilitate certification. Divide and conquer strategies for presenting the specification should promote reader comprehension. The specification produced as a result of the tangle phase should be written as a referential guide to the specification. Of course, the tangled output needs to be structured as required for processing by automated tools, but these constraints usually leave room for organizing the specification to improve accessibility. For example, the EVES theories that support the repeater assurance argument are modularized and indexed for ease of reference.

Do not use LP tools too early in the development.

The early stages of writing specifications for ease of understanding and verification involve a discovery period as the author experiments with different abstractions, e.g., specification models and data structures, in the search for the optimal solution. We discovered that the use of LP tools during this period interferes with the discovery process. More specifically, the slightly extended development process introduced by the tangle phase obstructs the trial and error nature of discovery. The general approach we adopted was to use the specification and verification tools without LP until the specification begins to stablize. As the author becomes comfortable with the specification, documenting the solution in terms that are intuitive to a reader may proceed using LP tools.

Simplified document maintenance justifies increased turnaround time.

The document describing the assurance argument necessarily involves many elements of the system specification. Maintaining the consistency of the argument with the actual system specification (as input to the specification and verification tools) is critical to the certification. Unfortunately, the natural evolution of the specification during both development and maintenance phases makes maintaining this consistency difficult even if the developer waits until the specification stablizes before beginning to document the argument. The common approach of maintaining this consistency manually is both time consuming and error-prone. LP techniques maintain consistency as a matter of course since both the argument and the specification are generated from a single source file. The price for this is an extended development process, which increases the time it takes to process changes to the specification. Nevertheless, the benefits of the LP approach in terms of simplified document maintenance significantly outweighs the disadvantages associated with increased turnaround time.

LP techniques support integrating formal and informal arguments.

A larger percentage of the repeater's assurance argument than expected was not amenable to formalization. Most of the argument's informality was due to assumptions that the CSP computational paradigm makes about the environment and implementation of CSP processes. These assumptions imply new critical requirements for the repeater that cannot be stated formally in the CSP model and must be verified informally. Tracing both formally and informally stated critical requirements through the levels of specification refinement is crucial to the cohesion of the assurance argument. The LP techniques and tools provided critical support for interleaving the informal and formal traces of requirements. We believe that the degree of informality of the repeater's argument was not peculiar to the repeater problem, but typical of nontrivial systems in general.

Document only high-level structure of machine-generated proofs.

Documenting a rigorous assurance argument requires deciding how best to present machine-generated proofs. Even sophisticated provers like EVES do not generate journal-level proofs. If cost were no issue, the ideal approach would be to provide intuitive descriptions of both the formal and informal parts of the assurance arguments. For the repeater argument, we chose a middle road: we provide an intuitive description of the informal parts and carefully outline the boundary of the formal parts.

The boundary of the formal parts of the repeater assurance argument was documented by describing only the high-level structure of the formal proofs, omitting the details about why the proofs hold. This approach permits covering all aspects of the argument once in detail and permits assessing the completeness of the argument. Unfortunately, it assumes the proof tools are sound, i.e., they cannot generate a proof of a false conjecture. Although proof tools are relatively reliable, tools that would help certifiers review the proofs produced by proof assistants would reduce the chances of basing an assurance argument on a false machine-generated proof.

Do not use LP tools as typesetters.

Many of the LP tools, including FunnelWeb, follow Knuth's original philosophy of providing enough LP-specific document formatting commands to produce nice documentation. The goal is to make knowledge of the underlying typesetter unnecessary to use LP tools. As the refinement of the repeater's assurance argument progressed, however, it became clear that the FunnelWeb formatting commands were insufficient by themselves to produce the documentation desired. Rather than mixing LaTeX and FunnelWeb formatting commands, we restructured the document using FunnelWeb's commands only for defining macros. All other document formatting was specified in LaTeX. This approach simplified the document construction process considerably.

Constructing assurance arguments requires increased support for referencing macros.

An assurance argument involves describing the relationships between entities of the specification and implementation in a more complicated way than that required for traditional literate programming. In particular, we found that FunnelWeb, when applied to assurance argument development, does not adequately support referencing macros. We often wanted to reference a macro definition by its name and definition number or to expand it "in-line" in the narrative (i.e., the non-code portion of the document). Because these references occur outside of a macro definition, FunnelWeb simply ignores them during the tangle and weave phases. Other members of the `WEB` family of tools may support such referencing and should be reviewed for future applications of LP to construct assurance arguments.

Methods are needed to integrate tools with LP.

While LP tools provide a useful function, they cannot, nor should they be expected to, provide all of the functions needed to document high assurance systems. For example, tools are needed to help trace both formally and informally specified requirements through the levels of refinement and for analyzing these traces for consistency and completeness. Other tools such as prettyprinters, proof review tools, graphical specification and simulation tools, and WYSIWYG editors could significantly facilitate the documentation and certification efforts. Methods are needed to organize and integrate such tools to best support the development of high assurance systems.

6 Conclusions

Our experience demonstrates the potential benefit of using LP tools and techniques in the development of understandable assurance arguments. In fact, the benefit of using LP to develop assurance arguments for trusted systems that require independent certification exceeds the benefit of using LP for traditional programming. The inevitable use of many different methods, both formal and informal, in the development of the assurance argument requires their application to be documented in a common framework to ensure coherence and readability. LP tools allow this framework to be constructed while maintaining its consistency with formal specifications that are input to specification and verification systems. This approach is more effort than documenting only the individual specifications, but it yields a more cohesive assurance argument and provides certifiers assurance that the running system conforms to its documentation.

The simple application of LP described in this paper is only a single case study, but the approach appears promising. We believe that its use would have prevented many of the problems that we encountered in the certification of the network security device [25]. Currently, we are using LP techniques in a larger scale, multi-member team development of a secure functional extension of the Navy's Joint Maritime Command Information System (JMCIS) [8]. We have defined the overall functional requirements and operation formally in the langauges of Statemate [9]. The Statemate specification provides a formal structure for the overall system assurance strategy and argument, which is being documented in hypertext (HTML). The refinement of this assurance argument for a security-critical component of the extended JMCIS architecture using EVES and FunnelWeb is underway.

The user community for LP tools is still very small, even after a decade of use. LP tools have not been commercialized (most are available free over the internet) and, thus, support is haphazard, at best. We hope that the benefit that accrues with the new application of LP described in this paper will revitalize the user community and stimulate commercial investment in the technology.

References

[1] Richard J. Auletta. Rapid-prototyping of high assurance systems. Technical report, George Mason University, Fairfax, Virginia, January 1993.

[2] Preston Briggs. Nuweb, A simple literate programming tool. Technical report, Rice University, Houston, TX, 1993.

[3] Marcus E. Brown and David Cordes. A literate programming design language. In *Proc. Comp. Euro 90, IEEE International Conference*, pages 548–549, Los Alamitos, CA, 1990. IEEE CS Press.

[4] Ricky W. Butler and George B. Finelli. The infeasibility of experimental quantification of life–critical software reliability. In *Proceedings of the ACM SIGSOFT Conference on Software for Critical Systems*, New Orleans, LA, December 1991. Also in *Software Engineering Notes*, Vol. 16, No. 5, December, 1991.

[5] Commission of the European Communities, Luxembourg. *Information Technology Security Evaluation Criteria (ITSEC)*, June 1991.

[6] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. Reference manual for the language Verdi. Technical Report TR-91-5429-09a, ORA Canada, Ottawa, Ontario, September 1991.

[7] Formal Systems (Europe) Ltd. *Failures Divergence Refinement: User Manual and Tutorial*, January 1994.

[8] Judith N. Froscher, David M. Goldschlag, Myong H. Kang, Carl E. Landwehr, Andrew P. Moore, Ira S. Moskowitz, and Charles N. Payne. Improving inter-enclare information flow for a secure strike planning application. In *Proc. 11th Annual Computer Security Applications Conference*, New Orleans, LA, December 1995. IEEE Computer Society Press.

[9] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[10] C. A. R. Hoare and J. C. Shepherdson, editors. *Mathematical Logic and Programming Languages.* Prentice-Hall, 1985.

[11] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[12] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2), May 1984.

[13] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting.* Addison-Wesley, Reading, MA, USA, 1986.

[14] Donald E. Knuth. *Literate Programming.* CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[15] Sentot Kromodimoeljo, Bill Pase, Mark Saaltink, Dan Craigen, and Irwin Meisels. EVES: An overview. Technical report, ORA Canada, Ottawa, Ontario, February 1993.

[16] Leslie Lamport. *A Document Preparation System LATEX: User's Guide and Reference Manual.* Addison-Wesley Publishing Company, 1986.

[17] Carl E. Landwehr. The RS-232 software repeater problem. Cipher Newsletter of the Technical Committee on Security and Privacy, Summer 1989.

[18] Silvio Levy. Literate programming and cweb. *Computer Language*, 10(1):67–68, 70, January 1993.

[19] David Mihelcic, Andrew Moore, and Charles Payne, Jr. If a tree falls in the woods ... In Chuck Howell, editor, *Notes from the MITRE Workshop on Assurance for Critical Software.* MITRE Corporation, September 10 and 11 1992.

[20] Andrew P. Moore. The specification and verified decomposition of system requirements using CSP. *IEEE Transactions on Software Engineering*, 16(9):932–948, September 1990.

[21] Andrew P. Moore. The EVES CSP library. NRL Technical Memorandum 5540-153:apm, Naval Research Laboratory, Washington, D.C., August 1994.

[22] Andrew P. Moore and Charles N. Payne, Jr. The RS-232 character repeater refinement and assurance argument. NRL Technical Memorandum 5540-034:amcp, Naval Research Laboratory, 1994.

[23] National Computer Security Center, Ft. Meade, MD. *DoD 5200.28-STD, Trusted Computer System Evaluation Criteria*, December 1985.

[24] Bill Pase and Sentot Kromodimoeljo. A user's guide to a skeletal CSP theory in EVES. Technical Report TR-92-5469-03, ORA Canada, Ottawa, Ontario, July 1992.

[25] Charles N. Payne, Jr., Andrew P. Moore, and David M. Mihelcic. An experience modeling critical requirements. In *Proc. COMPASS 94*, Gaithersburg, MD, June 1994. IEEE.

[26] Norman Ramsey. Weaving a language independent web. *Communications of the Association for Computing Machinery*, 32(9):1051–1055, September 1989.

[27] Trygve Reenskaug and Anne Lise Skaar. An environment for literate smalltalk programming. In *OOPSLA'89 Proceedings*, pages 337–345, 1989.

[28] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of CAR Hoare.* Prentice-Hall International, 1994.

[29] E. Wayne Sewell. How to MANGLE your software: the WEB system for Modula-2. *TUGboat*, 8(2):118–122, July 1987.

[30] Martin Simons, Maya Biersack, and Robert Raschke. Literate and structured presentation of formal proofs. In E.R. Olderog, editor, *IFIP Working Conference on Programming Concepts, Methods and Calculi*. North Holland, 1994.

[31] Ross Williams. FunnelWeb user's manual. Technical report, University of Adelaide, Adelaide, South Australia, Australia, May 1992. Available via anonymous ftp to ftp.adelaide.edu.au in /pub/funnelweb.

[32] Wai Wong. Mweb: Proof script management utilities. ftp.cl.cam.ac.uk:/contrib/mweb/manual, University of Cambridge, Computer Laboratory, Cambridge, CB23QG, England, 1994.

A Excerpt from Assurance Argument

This chapter specifies the critical requirements for a CSP process called `Rptr` that represents the repeater.[4]

.

.

.

Specification of the Formal Assertions

The CSP process `Rptr` has three parameters, which are constant for any particular instantiation of `Rptr`:

- `chsz` represents the length in bits of a character processed. All characters processed by `Rptr` are delimited by a `startbit`/`stopbit` combination.
- `buffsz` represents the capacity in characters of the internal buffer.
- `tick` represents the event representing successful termination. We use the variable `tick`, by convention, to represent a special event that occurs automatically when (and only when) a process terminates successfully. `Rptr` terminates successfully after it halts due to receiving a character that causes an overflow; otherwise, `Rptr` continues to process characters that it receives.

⟨*Definition Stub of Rptr*⟩[1] ≡
```
    function Rptr(chsz,buffsz,tick);
```
This macro is invoked in definition 219.

.

.

.

[4]Although a thorough understanding of this example requires familiarity with CSP and EVES, the reader should be able to get a feel for the flow of an assurance argument using LP. See [22] for the complete repeater assurance argument, including an overview of the languages of CSP and EVES.

Top-Level Requirements Structure

Our goal is to specify formally as trace specifications the critical requirements for `Rptr`.[5] Assuming `valid_relay` is the name for these requirements, our specification is as follows:[6]

⟨*Rptr satisfies valid_relay*⟩[7]M ≡
```
    axiom Rep_sat_spec(chsz,buffsz,tick) =
    begin
             chsz >= 0
         and buffsz >= 0
         and not iscomm tick
       -> Rptr(chsz,buffsz,tick)
         sat valid_relay(⟨Rptr alphabet⟩[27],
                          chsz,buffsz,tick)
    end Rep_sat_spec;
```
This macro is invoked in definitions 219 and 221.

A trace specification like `valid_relay` is simply a set of traces. Elements of the set are chosen from the universe of possible traces of events and must conform to `valid_relay_spec`:[7]

⟨*valid_relay*⟩[8]M ≡
```
    zf function valid_relay(a,chsz,buffsz,tick) =
    begin
      { tr1 in a^*
          | valid_relay_spec(tr1,chsz,buffsz,tick) }
    end valid_relay;
```
This macro is invoked in definitions 219 and 221.

We split `valid_relay_spec` into two pieces: one when `Rptr` has successfully terminated (i.e., `tick` is the last event of `Rptr`'s trace) and one when it has not. Traces of `Rptr` that end with `tick` must satisfy `Rptr`'s post condition; traces of `Rptr` that do not end with `tick` must satisfy `Rptr`'s invariant. This forms a natural partition of `valid_relay` since we can say more about the requirements of `Rptr` when it has terminated. At termination we can say that all even parity characters received by `Rptr` before the buffer overflows have been successfully transmitted. Before termination, all we can say is that some prefix of those characters have been transmitted. Henceforth, the phrase *valid characters* refers only to those characters of even parity received before an overflow.[8]

[5]The repeater's critical requirements were described informally in Section 4.

[6]The predicate `iscomm` holds if its argument is a CSP communication event. `P sat S` holds if process `P` satisfies trace specification `S`. Viewing process `P` as the set of traces in which `P` may engage and `S` as the set of acceptable traces, the `sat` predicate is equivalent to the subset predicate, i.e., process `P` may engage only in traces allowed by `S`.

[7]The parameter `a` represents the alphabet of all possible repeater events; `a^*` represents all possible traces formed from these events.

[8]`outbit` is the name of the repeater's output port; `inbit` is the name of the input port.

⟨*Constraints on Rptr's trace*⟩[9]**M** ≡

```
function valid_relay_spec(tr1,chsz,buffsz,tick) =
begin
  if ⟨Rptr terminates⟩[10]
  then ⟨All valid characters were transmitted over outbit⟩[11]
  else ⟨A subsequence of valid characters were transmitted over outbit⟩[22]
  end if
end valid_relay_spec;
```

This macro is invoked in definitions 219 and 221.

⟨*Rptr terminates*⟩[10] ≡

```
    not null tr1
and last(tr1) = tick
```

This macro is invoked in definition 9.

Repeater Post Condition

Under the assumption that Rptr has terminated, the three critical requirements [9] taken together require that up to the point at which an overflow occurs, the output stream of characters must be identical to the input stream of characters with characters of odd parity removed. Also, nothing other than identifiable and complete characters may be transmitted over `outbit`.

⟨*All valid characters were transmitted over outbit*⟩[11] ≡

```
    ⟨Character sequence transmitted over outbit⟩[12]
  = ⟨Valid character sequence received over inbit⟩[13]
  and ⟨Only whole characters were transmitted over outbit⟩[21]
```

This macro is invoked in definition 9.

Character sequence transmitted over outbit

The sequence of bits traversing `outbit`, i.e., `tr1 |= outbit`, is a flat representation of the characters processed by `Rptr`. The following formats this bit sequence into the character sequence it represents:

⟨*Character sequence transmitted over outbit*⟩[12]**M** ≡

```
⟨Bits to chars⟩[122](‘tr1 |= outbit’)
```

This macro is invoked in definitions 11 and 22.

Valid character sequence received over inbit

`valid_input_chars` returns the sequence of even parity characters received over `inbit` in trace `tr1` before an overflow occurs. This sequence is calculated by restricting the bits received before an overflow occurs to those characters of even parity.[10]

⟨*Valid character sequence received over inbit*⟩[13]**M** ≡

```
valid_input_chars(tr1,chsz,buffsz,tick)
```

This macro is invoked in definitions 11 and 22.

⟨*Definition of valid_input_chars*⟩[14]**M** ≡

```
function valid_input_chars(tr1,chsz,buffsz,tick) =
begin
   ⟨Characters received over inbit before overflow⟩[15]
|^ ⟨Set of even parity characters⟩[133]
end valid_input_chars;
```

This macro is invoked in definitions 219 and 221.

The sequence of characters received over `inbit` before an overflow are derived by transforming to characters the sequence of bits received over `inbit` before an overflow.

⟨*Characters received over inbit before overflow*⟩[15] ≡

```
⟨Bits to chars⟩[122] (‘⟨Trace before overflow⟩[16] |= inbit’)
```

This macro is invoked in definition 14.

The trace before an overflow is derived by choosing the longest prefix of `tr1` for which the predicate `no_error_condition` holds. This operation is performed by the function `filter` of the `tr` library unit. Intuitively, `no_error_condition` describes the condition under which no overflow of the internal buffer has occurred.

⟨*Trace before overflow*⟩[16] ≡

```
tr!filter(tr1,⟨Rptr alphabet⟩[27],
          no_error_condition(⟨Rptr alphabet⟩[27],
                             chsz, buffsz))
```

This macro is invoked in definition 15.

Functions to be passed as parameters in SVerdi are represented as a set of ordered pairs where the first elements of the pairs form the domain and the second elements form the range.[11] Since `no_error_condition` is a boolean function, we define it as a set of ordered pairs with domain equal to the set of traces of the alphabet passed in and the range equal to the set of Boolean values. Each trace is mapped to the value returned when passed to the predicate `no_over_flow`.

⟨*Definition of no_error_condition*⟩[17]**M** ≡

```
zf function no_error_condition(a, chsz, buffsz) =
begin
  { -<tr1, no_over_flow(tr1, chsz, buffsz)>-
      | tr1 in a^* }
end no_error_condition;
```

This macro is invoked in definitions 219 and 221.

An overflow occurs when the difference between the number of even parity characters received over `inbit` and the number of characters transmitted over `outbit` exceeds `buffsz + 1`, at any point during execution. `buffsz` is incremented by 1 since `Rptr` may

[9]See Section 4 of this paper for the informal statement of these requirements.

[10]The operator `|^` represents the restriction operator, i.e., `t |^ A` returns trace `t` with elements not in set `A` removed.

[11]The delimiters for ordered pairs are denoted by `-<` and `>-`.

be processing a character in addition to the `buffsz` characters possibly held by the internal buffer.[12]

⟨*Definition of no_over_flow*⟩[18]**M** ≡

```
     function no_over_flow(tr1, chsz, buffsz) =
     begin
       all tr2:
           tr2 .<=. tr1
        ->     (len ⟨Sequence of even parity inbit chars over
tr2⟩[19])
              - (len ⟨Sequence of outbit chars over tr2⟩[20])
           <= buffsz + 1
     end no_over_flow;
```

This macro is invoked in definitions 219 and 221.

⟨*Sequence of even parity inbit chars over tr2*⟩[19]**M** ≡

```
     ⟨Bits to chars⟩[122](`tr2 |= inbit')
     |^ ⟨Set of even parity characters⟩[133]
```

This macro is invoked in definitions 18 and 33.

⟨*Sequence of outbit chars over tr2*⟩[20]**M** ≡

```
     ⟨Bits to chars⟩[122](`tr2 |= outbit')
```

This macro is invoked in definition 18.

Only whole characters were transmitted over outbit

Specifying that no spurious bits were transmitted is a simple matter of stating that the sequence of bits that traversed `outbit` is a multiple of the character size, `chsz`, plus 2 for the start and stop bits delimiting each character transmitted.

⟨*Only whole characters were transmitted over outbit*⟩[21] ≡

```
     (len tr1 |= outbit) mod (chsz + 2) = 0
```

This macro is invoked in definition 11.

Repeater Invariant

Before `Rptr` terminates, we cannot guarantee that every even parity character received has been transmitted. We can guarantee, however, that the sequence of characters transmitted over `outbit` must be a prefix of the sequence of even parity characters received over `inbit` before the overflow occurred. This is easily specified in terms of the primitives already defined.

⟨*A subsequence of valid characters were transmitted over outbit*⟩[22] ≡

```
     ⟨Character sequence transmitted over outbit⟩[12]
       .<=. ⟨Valid character sequence received over inbit⟩[13]
```

This macro is invoked in definition 9.

.

.

.

[12] `.<=.` is the sequence prefix predicate.